Лекция 1. Основные понятия ООП

**Недостатки процедурного программирования**

1. Основным недостатком является неограниченный доступ к глобальным данным.

Большое количество связей усложняет разработку программы, т. к. имеется необходимость согласования работы всех функций при работе с определённой глобальной переменной. Возникают трудности при поиске функции, которые не корректно модифицируют данные. Сложно вносить изменения .

2. Другой проблемой является неадекватное отображение объектов реального мира, т. к. объект характеризуется не только определёнными свойствами, но и поведением.

**Объектно-ориентированный подход**

ООП не изменяет процесс выполнения программы, а является только другим способом её реализации.

Основополагающей идеей этого подхода является объединение свойств и поведения одного объекта. Данные объекта называются **полями**, а функции, принадлежащие объекту и, как правило, предназначенные для доступа к данным того объекта, называются **методами**. Прямой доступ к данным объекта из внешних функций, как правило, запрещён! Связывание в единое целое данных объекта и функций доступа к ним с целью создания объектов нового типа называется **инкапсуляцией (encapsulation)**. Описание инкапсулированного объекта называется **классом**. При объявлении класса выделение памяти не происходит (объекты этого класса не создаются). Все объекты, для которых выделена память, называются **экземплярами (объектами) класса**.

Использование свойств ранее определённых классов при образовании иерархии производных классов называется **наследованием (inheritance)**. Класс, который порождает все остальные классы, называется **базовым**. Классы, которые наследуют свойства базового класса, называются **производными** классами.

Если существующая операция наделяется возможностью совершения действий над операндами нового типа, то она называется **перегруженной**.

Возможность ассоциации некоторого имени с множеством уникальных для каждого уровня иерархии производных объектов называется **полиморфизмом (polymorphizm)**. Св-ва полиморфизма позволяют использовать одинаковое название метода для решения сходных, но несколько отличающихся у разных родственных классов задач. Обеспечивается это тем, что в классе "потомки" метод перегружается (переписывается по новому алгоритму). В результате в объекте-родителе и объекте-потомке будут существовать 2 метода, имеющие различные алгоритмы.

**Понятие функции**

**Функция** - это последовательность операторов, оформленная таким образом, что её можно вызвать из любого места программы.

тип\_возвр\_знач имя\_ф-ии (список\_параметров)

{

Тело функции

}

Массив нельзя передавать или возвращать.

**Передача параметров**

Как правило, помимо описания функций в программу помещается **прототип** - предварительное описание функции.

**- по значению**

int fun (double, int, char);//Прототип

int fun (double a, int b, char c)

{

Тело функции

}

…

r=fun (d, 8, chr);

В момент обращения к функции в памяти создаются переменные a, b, c соответствующих типов. В них копируются стоящие в заданном порядке данные, указанные при вызове функции. Связь между формальными и фактическими параметрами теряется.

Достоинства:

1) Любые изменения формальных параметров не влекут изменение фактических параметров в вызывающей функции.

2) В качестве параметров можно использовать не только переменные, но и константы, и выражения соответствующего типа.

**- по ссылке**

void fun (double &, int &);//Прототип

void fun (double &a, int &b)

{

Тело функции

}

…

fun (d, r);

При таком способе передачи передаётся не значение переменной, а её адрес, который ассоциируется с соответствующим параметром функции.

Достоинства:

1) Нет затрат ресурсов (памяти и процессорного времени) на создание формальных параметров.

2) Возможна передача результатов в вызывающую функцию.

Недостаток:

Возможность несанкционированного изменения данных в вызывающей функции. Для исключения изменения данных можно перед типом параметра указать ключевое слово const.

void fun (double &a, const int&b)

**- по указателю**

void fun (double \*, int \*);//Прототип

void fun (double \*a, int \*b)

{

Тело функции

}

…

fun (&f, &k);

Вместо операции разадресации используется операция косвенной адресации. При использовании формальных параметров внутри функции необходимо использовать операцию разадресации.

Аргументы, задаваемые по умолчанию, должны располагаться правее обычных аргументов.

Пример.

void fun (double, int b = 3, double h= 0.1);//Прототип

void fun (double a, int b, double h)

{

Тело функции

}

…

fun(d)//a=d; b=3; h=0.1;

fun(d, r)//a=d; b=r; h=0.1;

fun(d, r, f)//a=d; b=r; h=f;

Пример. Передать в функцию значения 2, 5, 8, используя способы передачи параметров по значению, по ссылке и по указателю. Получить результат, используя ссылочный указатель.

sum (int a, int \*b, int &c, int &d)

{

d=a+\*b+c;

return;

}

int main()

{

int x=5, y=8, z;

sum(2, &x, y, z);

cout<<"sum="<<z<<endl;//+ очитка буфера

return 0;

}

**Передача переменного числа параметров**

тип\_возвр\_знач имя\_ф-ии (список\_параметров, …)

Список параметров должен содержать хотя бы 1 обязательный параметр.

**Макросы** находятся в библиотеке **stdarg**.

void va\_start (va\_list указатель, имя\_послед\_обязат\_аргумента)//Начинает работу со списком, устанавливает указатель на последний обязательный аргумент.

void va\_arg (va\_list указатель, тип\_аргумента)//Возвращает значение очередного необязательного аргумента из списка, и указатель сдвигается к след. аргументу. Макрос не контролирует достижение конца списка.

void va\_end (va\_list указатель)//Завершает работу со списком и освобождает память.

Пример. Посчитать сумму введённых аргументов.

#include <iostream>

#include <stdarg.h>

#include <conio.h>

using namespace std;

int fun(int,...);

int fun (int n,...)

{

int s=n, ar, k=n;

va\_list argm;

va\_start (argm, n);

ar=va\_arg (argm, int);

while (ar != k)

{

s+=ar;

ar=va\_arg (argm, int);

}

va\_end (argm);

return s;

}

int main()

{

cout<<"sum="<<fun(6,1,2,3,4,5,6)<<endl;//21

cout<<"sum="<<fun(3,1,2,3)<<endl;//6

\_getch();

return 0;

}

**Встраиваемые функции**

При использовании данных функций тело функций вносится непосредственно в текст программы во всех местах, где встречается вызов функции.

inline double fun (int a, double b)

{

return a\*b;

}

Особенности применения:

1) Желательно применять для небольших функций, так как вставленные тела функций значительно увеличивают размер программы.

2) Любое изменение функции inline требует перекомпиляции всех программ и её использующих.

Компилятор игнорирует спецификацию inline в случаях:

1) Наличие операторов циклов, переключателей (switch) и операторов безусловного перехода (goto).

2) Наличие оператора return в функция, не возвращающих значения.

3) В случае рекурсивного вызова функции.

4) При использовании статических переменных.

Перегрузка функций - это использование нескольких функций, имеющих одинаковое имя, но разное количество и типы параметров.

Пример. Написать программу, которая суммирует или 2, или 3 целых числа, или массив целых чисел.

#include <iostream.h>

int sum (int a, int b)

{

return a+b;

}

int sum (int a, int b, int c)

{

return a+b+c;

}

int sum (int mas[], int n)

{

int s=0;

for (int i=0; i<n; i++) s+=mas[i];

return s;

}

int main()

{

int a[4]={1, 2, 3, 4};

cout<<"sum="<<sum(1,2)<<endl;

cout<<"sum="<<sum(1,2,3)<<endl;

cout<<"sum="<<sum(a,4)<<endl;

return 0;

}

**Передача массивов в функции**

void funm(int[]);

…

void funm(int b[])

{

Тело функции

}

…

funm(a);//Вызов функции

Как правило, передаётся размер массива. При передаче двумерного массива для первой размерности скобки остаются пустыми, а для других константой указывается размер.

void funm(int[][3]);

…

void funm(int b[][3])

{

Тело функции

}

…

funm(a);//Вызов функции

**Указатель на функцию**

Имя функции является указателем на начало блоков в оперативной памяти, в которых размещается функция. Данный указатель является константой. При объявлении указателя на функцию он должен иметь тот же тип и те же аргументы, как и функция, на которую он указывает.

Пример. Имеется функция double y ( double x, int n).

Декларируем указатель на функцию: double(\*fun)(double, int);

Присвоим указателю fun адрес функции y: fun=y;

Функцию через указатель можно вызвать так:

x=fun(t,m);

Или так:

x=(\*fun)(t,m);

Указатель на функцию или массив не требует операции разадресации.

Пример. Вывести на экран таблицу значений функции Y(x)=sinx и её разложения в ряд S(x)= x - (x^3)/3! +…+(-1)\*(x^(2\*n+1))/(2\*n+1)!, n=20.

#include <iostream.h>

#include <math.h>

#include <iomanip.h>

double y (double x){

return sin(x);}

double s (double x){

double sum=x, a=x;

for (int i=13; i<20; i++){

a\*=pow(x,2)/(2\*i\*(2\*i+1));

sum+=a;

}

return sum;

}

typedef double (\*uf)(double);

void tabl (double a, double b, double h, uf fun){

int k=0, double sum;

for (double x=a; x<=b; x+=h)

cout<<setw(8)<<x<<setw(15)<<fun(x)<<endl;

}

int main()

{

cout<<setw(8)<<"x"<<setw(15)<<"y(x)"<<endl;

tabl (0.1, 0.8, 0.1, y);

cout<<endl;

cout<<setw(8)<<"x"<<setw(15)<<"s(x)"<<endl;

tabl (0.1, 0.8, 0.1, s);

return 0;

}